Attachment D

Pages MU0023213, MU0023250-52, MU0023254-55, MU0023259, and MU0023308-14 of Exhibit A2 included with the Declaration of Craig Hansen Under 37 CFR § 1.313 filed on September 18, 2009

(14 pages)

microunity

Terpsichore System Architecture

Copy Number: 247 **REDACTED**

Issued To: _____
Final Test

Issued By: _____
(MicroUnity officer or director)

Craig Hansen
Chief Architect
MicroUnity Systems Engineering, Inc.
255 Caspian Drive
Sunnyvale, CA 94089-1015
Tel: (408) 734-8100 Fax: (408) 734-8136
EMail: craig@microunity.com

callee (non-leaf):

```
S.64        sp,off(dp)
L.64        sp,off(dp)
S.64        link,off(sp)
S.64        dp,off(sp)
... (using dp)
L.64        link,off(sp)
L.64        dp,off(sp)
L.64        sp,off(dp)
B.DOWN      link
```

callee (leaf):

```
... (using dp)
B.DOWN      link
```

The callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, except as a region to receive parameters held in memory.

Pipeline Organization

Terpsichore performs all instructions as if executed one-by-one, in-order, with precise exceptions always available. Consequently, code which ignores the subsequent discussion of Terpsichore pipeline implementations will still perform correctly. However, the highest performance of the Terpsichore processor is achieved only by matching the ordering of instructions to the characteristics of the pipeline. In the following discussion, the general characteristics of all Terpsichore implementations precedes discussion of specific choices for specific implementations.

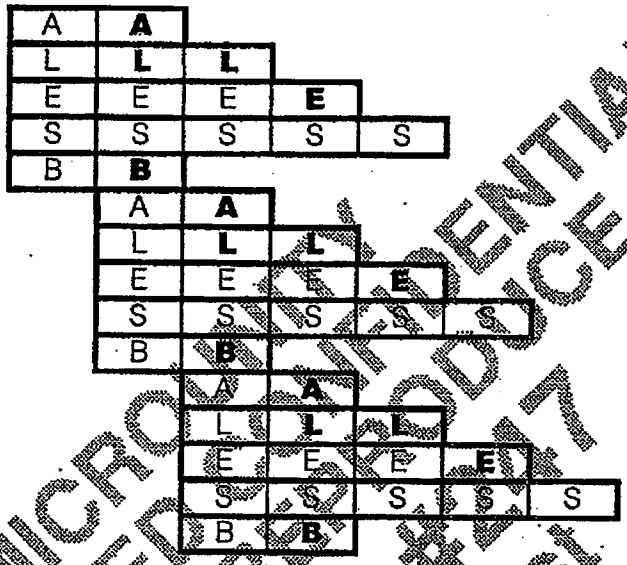
Super-string Pipeline

Terpsichore is designed to fetch and execute several instructions in each clock cycle. For a particular ordering of instruction types, one instruction of each type may be issued in a single clock cycle. The ordering required is A, L, E, S, B; in other words, a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store, and a branch. Because of the organization of the pipeline, each of these instructions may be serially dependent. Instructions of type E include the fixed-point execute-phase instructions as well as floating-point and digital signal processing instructions. We call this form of pipeline organization "super-string,"[4] because of the ability to issue a string of dependent instructions in a single clock cycle, as distinguished from super-scalar or super-pipelined organizations, which can only issue sets of independent instructions.

These instructions take from two to five cycles of latency to execute, and a branch prediction mechanism is used to keep the pipeline filled. The diagram below shows a box for the interval between issue of each instruction and the completion.

[4]Readers with a background in theoretical physics may have seen this term in an other, unrelated, context.

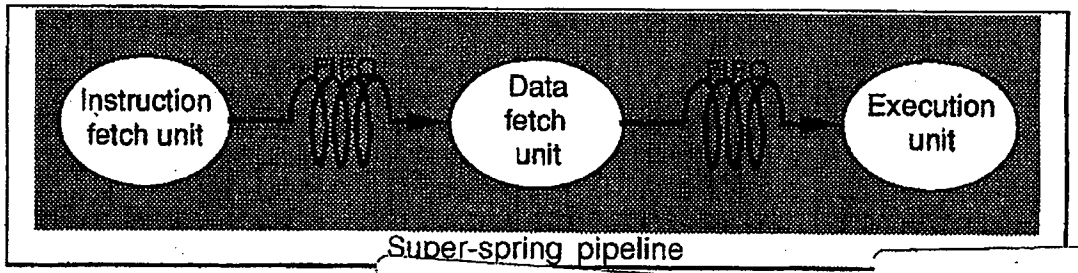
Bold letters mark the critical latency paths of the instructions, that is, the periods between the required availability of the source registers and the earliest availability of the result registers. The A-L critical latency path is a special case, in which the result of the A instruction may be used as the base register of the L instruction without penalty. E instructions may require additional cycles of latency for certain operations, such as fixed-point multiply and divide, floating-point and digital signal processing operations.



Super-spring Pipeline

Terpsichore provides an additional refinement to the organization defined above, in which the time permitted by the pipeline to service load operations may be flexibly extended. Thus, the front of the pipeline, in which A, L and B type instructions are handled, is decoupled from the back of the pipeline, in which E, and S type instructions are handled. This decoupling occurs at the point at which the data cache and its backing memory are referenced; similarly, a FIFO that is filled by the instruction fetch unit decouples instruction cache references from the front of the pipeline shown above. The depth of the FIFO structures is implementation-dependent, i.e. not fixed by the architecture.

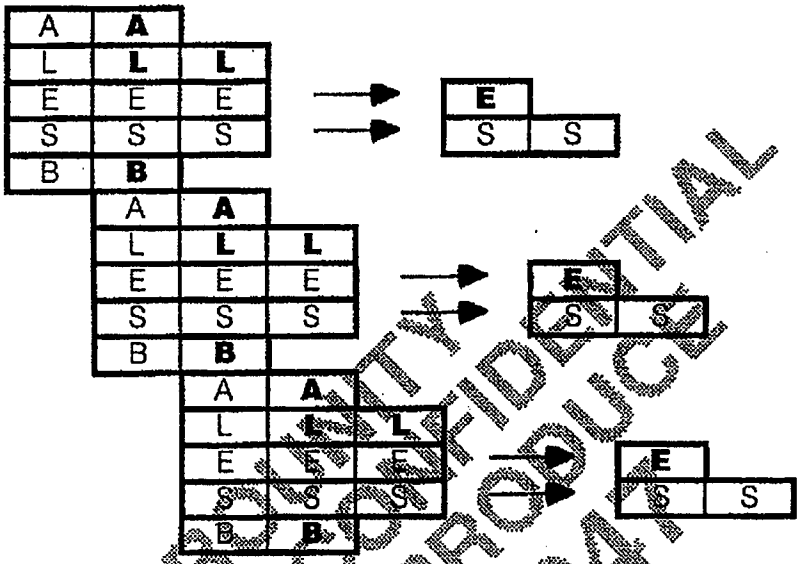
The diagram below indicates why we call this pipeline organization feature "super-spring," an extension of our super-string organization.



Super-spring pipeline

MU 0023251

With the super-spring organization, the latency of load instructions can be extended, so execute instructions are deferred until the results of the load are available. Nevertheless, the execution unit still processes instructions in normal order, and provides precise exceptions.



Branch/fetch Prediction

Terpsichore does not have delayed branch instructions, and so relies upon branch or fetch prediction to keep the pipeline full around unconditional and conditional branch instructions. The hardware prediction mechanism is tuned for optimizing conditional branches that close loops or express frequent alternatives, and will generally require substantially more cycles when executing conditional branches whose outcome is not predominately taken or not-taken. For such cases, the use of code which avoids conditional branches in favor of the use of set on compare and multiplex instructions may result in greater performance.

MU 0023252

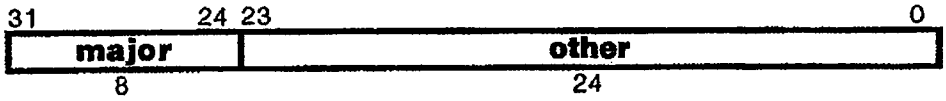
Additional Load and Execute Resources

Studies of the dynamic distribution of Terpsichore instructions on the various benchmark suites indicate that the most frequently-issued instruction classes are load instructions and execute instructions. In a high-performance Terpsichore implementation, it is advantageous to consider execution pipelines in which the ability to target the machine resources toward issuing load and execute instructions is increased.

One of the means to increase the ability to issue execute-class instructions is to provide the means to issue two execute instructions in a single-issue string. The execution unit actually requires several distinct resources, so by partitioning these resources, the issue capability can be increased without increasing the number of functional units, other than the increased register file read and write ports. The partitioning favored for the initial implementation places all instructions that

Instruction Set

All instructions are 32 bits in size, and use the high order 8 bits to specify a major operation code.

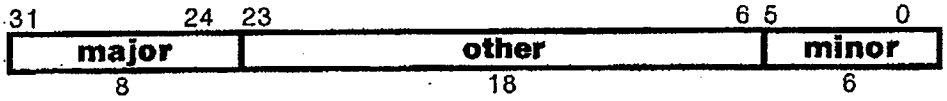


The major field is filled with a value specified by the following table:

MAJOR	0	32	64	96	128	160	192	224
0	ARES	ESETIE	FMULADD16	GMULADD1	LU16LAI	SAA64LAI	BFE16	BE
1		ESETINE	FMULADD32	GMULADD2	LU16BAI	SAA64BAI	BFNE16	BNE
2		ESETIL	FMULADD64	GMULADD4	LU16LI	SCAS64LAI	BFUE16	BL
3		ESETIGE	FMULADD128	GMULADD8	LU16BI	SCAS64BAI	BFNUE16	BGE
4	AADDI	EADDI	FMULSUB16	GMULADD16	LU32LAI	SMAS64LAI	BFNUGE16	
5		EADDIO	FMULSUB32	GMULADD32	LU32BAI	SMAS64BAI	BFUGE16	
6		ESETIUL	FMULSUB64	GMULADD64	LU32LI	SMUX64LAI	BFUL16	BUL
7		ESETIUGE	FMULSUB128		LU32BI	SMUX64BAI	BFNUL16	BUGE
8		ESUBIE			L16LAI	S16LAI	BFE32	BANDE
9		ESUBINE		GUMULADD2	L16BAI	S16BAI	BFNE32	BANDNE
10		ESUBIL		GUMULADD4	L16LI	S16LI	BFUE32	BANDL
11		ESUBIGE		GUMULADD8	L16BI	S16BI	BFNUE32	BANDGE
12	ASUBI	ESUBIL	F.16	GUMULADD16	L32LAI	S32LAI	BFNUGE32	
13		ESUBIUGE	F.32	GUMULADD32	L32BAI	S32BAI	BFUGE32	
14		ESUBIBL	F.64	GUMULADD64	L32LI	S32LI	BFUL32	BANDG
15		ESUBIUGE	F.128		L32BI	S32BI	BFNUL32	BANDLE
16	AANDI	EANDI	GFMULADD16	G.EXTRACT.128	L64LAI	S64LAI	BFE64	
17	AORI	EORI	GFMULADD32	G.COMPOWER.ML	L64BAI	S64BAI	BFNE64	
18	AXORI	EXORI	GFMULADD64	GPOWER	L64LI	S64LI	BFUE64	
19		EMUX		GMUX	L64BI	S64BI	BFNUE64	
20	ANANDI	ENANDI	GFMULSUB16	G.EXTRACT.1	L128LAI	S128LAI	BFNUGE64	
21	ANORI	ENORI	GFMULSUB32	G.EXTRACT.64	L128BAI	S128BAI	BFUGE64	BGATEI
22		EADDIO	GFMULSUB64	G.EXTRACT	L128LI	S128LI	BFUL64	
23		ESUBIUO		F.128	L128BI	S128BI	BFNUL64	
24			F.1	G.1	L8I	S8I	BFE128	
25			F.2	G.2	LU8I		BFNE128	
26			F.4	G.4			BFUE128	
27			F.8	G.8			BFNUE128	
28	ACOPYI	ECOPYI	GF.16	G.16	BGATEI		BFNUGE128	BI
29			GF.32	G.32			BFUGE128	BLINKI
30			GF.64	G.64			BFUL128	
31	A.MINOR	E.MINOR		G.MINOR	L.MINOR	S.MINOR	BFNUL128	B.MINOR

major operation code field values

For the major operation field values A.MINOR, L.MINOR, E.MINOR, F.16, F.32, F.64, F.128, GF.16, GF.32, GF.64, G.1, G.2, G.4, G.8, G.16, G.32, G.64, S.MINOR and B.MINOR, the lowest-order six bits in the instruction specify a minor operation code:



[5]Blank table entries cause the Reserved Instruction exception to occur.

MU 0023254

The minor field is filled with a value from one of the following tables:

A.MINOR	0	8	16	24	32	40	48	56
0			AAND					
1			AOR					
2			AXOR					
3			AANDN					
4	AADD	ASUB	ANAND					ASHLI
5			ANOR					
6			AXNOR					ASHRI
7			AORN					AUSHRI

minor operation code field values for A.MINOR

E.MINOR	0	8	16	24	32	40	48	56
0	ESETE	ESUBE	EAND		EALMS			
1	ESETNE	ESUBNE	EOR		EASUM			
2	ESETL	ESUBL	EXOR	EEXPAND				EEXPANDI
3	ESETGE	ESUBGE	EANDN	EUEXPAND				EUEXPANDI
4	EADD	ESUB	ENAND	ESHL				ESHLI
5	EADDSO	ESUBSO	ENOR					
6	ESETUL	ESUBUL	EXNOR	ESHR	E.GATHER			ESHRI
7	ESETUGE	ESUBUGE	EORN	EUSHR	E.SCATTER			EUSHRI

minor operation code field values for E.MINOR

F.size	0	8	16	24	32	40	48	56
0	FADD.N	FADD.T	FADD.F	FADD.C	FADD	FADD.X	FSETE	FSETE.X
1	FSUB.N	FSUB.T	FSUB.F	FSUB.C	FSUB	FSUB.X	FSETNE	FSETNE.X
2	FMUL.N	FMUL.T	FMUL.F	FMUL.C	FMUL	FMUL.X	FSETUE	FSETUE.X
3	FDIV.N	FDIV.T	FDIV.F	FDIV.C	FDIV	FDIV.X	FSETNUE	FSETNUE.X
4	F.UNARY.N	F.UNARY.T	F.UNARY.F	F.UNARY.C	F.UNARY	F.UNARY.X	FSETNUGE	FSETL.X
5							FSETUGE	FSETNL.X
6							FSETUL	FSETNGE.X
7							FSETNUL	FSETGE.X

minor operation code field values for F.size

GF.size	0	8	16	24	32	40	48	56
0	GFADD.N	GFADD.T	GFADD.F	GFADD.C	GFADD	GFADD.X	GFSETE	GFSETE.X
1	GFSUB.N	GFSUB.T	GFSUB.F	GFSUB.C	GFSUB	GFSUB.X	GFSETNE	GFSETNE.X
2	GFMUL.N	GFMUL.T	GFMUL.F	GFMUL.C	GFMUL	GFMUL.X	GFSETUE	GFSETUE.X
3	GFDIV.N	GFDIV.T	GFDIV.F	GFDIV.C	GFDIV	GFDIV.X	GFSETNUE	GFSETNUE.X
4	GF.UNARY.N	GF.UNARY.T	GF.UNARY.F	GF.UNARY.C	GF.UNARY	GF.UNARY.X	GFSETNUGE	GFSETL.X
5							GFSETUGE	GFSETNL.X
6							GFSETUL	GFSETNGE.X
7							GFSETNUL	GFSETGE.X

minor operation code field values for GF.size

G.size	0	8	16	24	32	40	48	56
0	GSETE		GAND		GCOPY		GMUL	
1	GSETNE		GOR		GSWAP		GUMUL	GCOMPRESSI
2	GSETL		GXOR	GEXPAND			GDIV	GEXPANDI
3	GSETGE		GANDN	GUEXPAND	GSHUFFLE		GUDIV	GUEXPANDI
4	GADD	GSUB	GNAND	✓ GSHL				✓ GSHLI
5			GNOR		G.EXTRACT			
6	GSETUL		GXNOR	✓ GSHR	G.GATHER			✓ GSHRI
7	GSETUGE		GORN	✓ GUSHR	G.SCATTER			✓ GUSHRI

minor operation code field values for G.size

```
                FloatingPoint(minor.op, major.size, minor.round, ra, rb, rc)
            F.UNARY.N, F.UNARY.T, F.UNARY.F, F.UNARY.C,
            F.UNARY, F.UNARY.X:
                case unary of
                    F.ABS, F.NEG, F.SQR,
                    F.HALF, F.SINGLE, F.DOUBLE, F.QUAD,
                    F.INT, F.FLOAT:
                        FloatingPointUnary(unary.op, major.size, minor.round,
                            ra, rc)
                    others:
                        raise ReservedInstruction
                endcase
            others:
                raise ReservedInstruction
        endcase
    GMULADD1, GMULADD2, GMULADD4,
    GMULADD8, GMULADD16, GMULADD32,
    GUMULADD2, GUMULADD4,
    GUMULADD8, GUMULADD16, GUMULADD32,
    GMUX, GMUXGATHER, GSCATTERMUX, G.EXTRACT.128:
        GroupTernary(major,size,ra,rb,rc,rd)
    G.EXTRACT.I, G.EXTRACT.I64:
        GroupExtractImmediate(major,ra,rb,rc,minor)
    G.1, G.2, G.4, G.8, G.16, G.32:
        case minor of
            G.SHL, G.SHR, G.USHR, G.ADD, G.SUB, G.MUL, G.UMUL,
            G.AND, G.OR, G.XOR, G.ANDN, G.NAND, G.NOR, G.XNOR, G.ORN,
            G.SET.E, G.SET.NE, G.SET.L, G.SET.GE, G.SET.UL, G.SET.UGE,
            G.COPY, G.SWAP, G.DEAL, G.SHUFFLE, G.COMPRESS, G.EXPAND,
            G.GATHER, G.SCATTER:
                Group(minor,major,ra,rb,rc)
            G.COMPRESS.I, G.EXPAND.I, G.SHL.I, G.SHR.I, G.U.SHR.I:
                GroupShortImmediate(minor,major,ra,simm,rc)
            G.EXTRACT.I:
                GroupExtractImmediate(major,ra,rb,rc,minor)
            others:
                raise ReservedInstruction
        endcase
    GFMULADD16, GFMULADD32, GFMULADD64,
    GFMULSUB16, GFMULSUB32, GFMULSUB64:
        GroupFloatingPointTernary(major,ra,rb,rc,rd)
    GF.16, GF.32, GF.64, GF.128:
        case minor of
            GF.ADD.N, GF.SUB.N, GF.MUL.N, GF.DIV.N,
            GF.ADD.T, GF.SUB.T, GF.MUL.T, GF.DIV.T,
            GF.ADD.F, GF.SUB.F, GF.MUL.F, GF.DIV.F,
            GF.ADD.C, GF.SUB.C, GF.MUL.C, GF.DIV.C,
            GF.ADD, GF.SUB, GF.MUL, GF.DIV,
            GF.ADD.X, GF.SUB.X, GF.MUL.X, GF.DIV.X,
            GF.SET.E, GF.SET.NE, GF.SET.UE, GF.SET.NUE,
            GF.SET.NUGE, GF.SET.UGE, GF.SET.UL, GF.SET.NUL,
            GF.SET.E.X, GF.SET.NE.X, GF.SET.UE.X, GF.SET.NUE.X,
            GF.SET.L.X, GF.SET.NL.X, GF.SET.NGE.X, GF.SET.GE.X:
                GroupFloatingPoint(minor.op, major.size, minor.round, ra, rb, rc)
            GF.UNARY.N, GF.UNARY.T, GF.UNARY.F, GF.UNARY.C,
            GF.UNARY, GF.UNARY.X:
                case unary of
                    GF.ABS, GF.NEG, GF.SQR,
```

Group

These operations take two values from a pair of registers, perform operations on groups of bits in the operands, and place the concatenated results in a register .

Operation codes

G.ADD.2	Group add pecks
G.ADD.4	Group add nibbles
G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.AND[10]	Group and
G.ANDN[11]	Group and not
G.COMPRESS.1	Group compress bits
G.COMPRESS.2	Group compress pecks
G.COMPRESS.4	Group compress nibbles
G.COMPRESS.8	Group compress bytes
G.COMPRESS.16	Group compress doublets
G.COMPRESS.32	Group compress quadlets
G.COMPRESS.64	Group compress octlets
G.COPY.1	Group copy bits
G.COPY.2	Group copy pecks
G.COPY.4	Group copy nibbles
G.COPY.8	Group copy bytes
G.COPY.16	Group copy doublets
G.COPY.32	Group copy quadlets
G.COPY.64	Group copy octlets
G.DEAL.1	Group deal bits
G.DEAL.2	Group deal pecks
G.DEAL.4	Group deal nibbles
G.DEAL.8	Group deal bytes
G.DEAL.16	Group deal doublets
G.DEAL.32	Group deal quadlets
G.DIV.64	Group signed divide octlets
G.EXPAND.1	Group signed expand bits
G.EXPAND.2	Group signed expand pecks
G.EXPAND.4	Group signed expand nibbles
G.EXPAND.8	Group signed expand bytes
G.EXPAND.16	Group signed expand doublets
G.EXPAND.32	Group signed expand quadlets
G.EXPAND.64	Group signed expand octlet

MU 0023308

[10]G.AND does not require a size specification, and is encoded as G.AND.1.
[11]G.ANDN does not require a size specification, and is encoded as G.ANDN.1. G.ANDN is used as the encoding for G.SET.L.1, and by reversing the operands, for G.SET.UL.1.

G.GATHER.2	Group gather pecks
G.GATHER.4	Group gather nibbles
G.GATHER.8	Group gather bytes
G.GATHER.16	Group gather doublets
G.GATHER.32	Group gather quadlets
G.GATHER.64	Group gather octlets
G.GATHER.128[12]	Group gather hexlets
G.MUL.1[13]	Group signed multiply bits
G.MUL.2	Group signed multiply pecks
G.MUL.4	Group signed multiply nibbles
G.MUL.8	Group signed multiply bytes
G.MUL.16	Group signed multiply doublets
G.MUL.32	Group signed multiply quadlets
G.MUL.64	Group signed multiply octlets
G.NAND[14]	Group nand
G.NOR[15]	Group nor
G.OR[16]	Group or
G.ORN[17]	Group or not
G.POLY.1	Group polynomial divide bits
G.POLY.2	Group polynomial divide pecks
G.POLY.4	Group polynomial divide nibbles
G.POLY.8	Group polynomial divide bytes
G.POLY.16	Group polynomial divide doublets
G.POLY.32	Group polynomial divide quadlets
G.POLY.64	Group polynomial divide octlets
G.SCATTER.2	Group scatter pecks
G.SCATTER.4	Group scatter nibbles
G.SCATTER.8	Group scatter bytes
G.SCATTER.16	Group scatter doublets
G.SCATTER.32	Group scatter quadlets
G.SCATTER.64	Group scatter octlets
G.SCATTER.128[18]	Group scatter hexlet
G.SHL.2	Group shift left pecks
G.SHL.4	Group shift left nibbles
G.SHL.8	Group shift left bytes
G.SHL.16	Group shift left doublets
G.SHL.32	Group shift left quadlets
G.SHL.64	Group shift left octlets

[12]G.GATHER.128 is encoded as G.GATHER.1

[13]G.MUL.1 is used as the encoding for G.UMUL.1.

[14]G.NAND does not require a size specification, and is encoded as G.NAND.1.

[15]G.NOR does not require a size specification, and is encoded as G.NOR.1.

[16]G.OR does not require a size specification, and is encoded as G.OR.1.

[17]G.ORN does not require a size specification, and is encoded as G.ORN.1. G.ORN is used as the encoding for G.SET.UGE.1, and by reversing the operands, for G.SET.GE.1.

[18]G.SCATTER.128 is encoded as G.SCATTER.1

G.SHR.2	Group signed shift right pecks
G.SHR.4	Group signed shift right nibbles
G.SHR.8	Group signed shift right bytes
G.SHR.16	Group signed shift right doublets
G.SHR.32	Group signed shift right quadlets
G.SHR.64	Group signed shift right octlets
G.SHUFFLE.1	Group shuffle bits
G.SHUFFLE.2	Group shuffle pecks
G.SHUFFLE.4	Group shuffle nibbles
G.SHUFFLE.8	Group shuffle bytes
G.SHUFFLE.16	Group shuffle doublets
G.SHUFFLE.32	Group shuffle quadlets
G.SWAP.1	Group swap bits
G.SWAP.2	Group swap pecks
G.SWAP.4	Group swap nibbles
G.SWAP.8	Group swap bytes
G.SWAP.16	Group swap doublets
G.SWAP.32	Group swap quadlets
G.U.DIV.64	Group signed divide octlets
G.U.EXPAND.1	Group unsigned expand bits
G.U.EXPAND.2	Group unsigned expand pecks
G.U.EXPAND.4	Group unsigned expand nibbles
G.U.EXPAND.8	Group unsigned expand bytes
G.U.EXPAND.16	Group unsigned expand doublets
G.U.EXPAND.32	Group unsigned expand quadlets
G.U.EXPAND.64	Group unsigned expand octlet
G.U.MUL.2	Group unsigned multiply pecks
G.U.MUL.4	Group unsigned multiply nibbles
G.U.MUL.8	Group unsigned multiply bytes
G.U.MUL.16	Group unsigned multiply doublets
G.U.MUL.32	Group unsigned multiply quadlets
G.U.MUL.64	Group unsigned multiply octlets
G.U.SHR.2	Group unsigned shift right pecks
G.U.SHR.4	Group unsigned shift right nibbles
G.U.SHR.8	Group unsigned shift right bytes
G.U.SHR.16	Group unsigned shift right doublets
G.U.SHR.32	Group unsigned shift right quadlets
G.U.SHR.64	Group unsigned shift right octlets
G.XNOR[19]	Group exclusive-nor
G.XOR[20]	Group exclusive-or

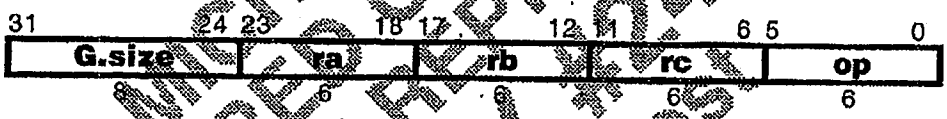
[19]G.XNOR does not require a size specification, and is encoded as G.XNOR.1. G.XNOR is used as the encoding for G.SET.E.1.

[20]G.XOR does not require a size specification, and is encoded as G.XOR.1. G.XOR is used as the encoding for G.ADD.1, G.SUB.1 and G.SET.NE.1.

class	op	size
linear	ADD	2 4 8 16 32 64
bitwise	AND ANDN NAND NOR OR ORN XNOR XOR	
signed multiply	MUL	1 2 4 8 16 32 64
unsigned multiply	U.MUL	2 4 8 16 32 64
signed divide	DIV	64
unsigned divide	U.DIV	64
rearrange	COPY DEAL SWAP SHUFFLE	1 2 4 8 16 32
	GATHER SCATTER	2 4 8 16 32 64
galois field	POLY	1 2 4 8 16 32 64
precision	COMPRESS EXPAND U.EXPAND	1 2 4 8 16 32 64
shift	SHL SHR U.SHR	2 4 8 16 32 64

Format

G.op.size rc=ra,rb



Description

Two values are taken from the contents of registers ra and rb. The specified operation is performed, and the result is placed in register rc.

Definition

```
def Group(op,size,ra,rb,rc)
    case op of
        G.MUL, G.U.MUL, G.DIV, G.U.DIV:
            a ← REG[ra]
            b ← REG[rb]
        G.ADD, G.SUB, G.SET.L, G.SET.UL, G.SET.E, G.SET.NE, G.SET.GE, G.SET.UGE,
        G.AND, G.OR, G.XOR, G.ANDN, G.NAND, G.NOR, G.XNOR, G.ORN,
        G.GATHER, G.SCATTER:
            a ← REG[ra]
            b ← REG[rb]
        G.COMPRESS, G.SHL, G.SHR, G.U.SHR, G.POLY:
            a ← REG[ra]
            b ← REG[rb]
        G.EXPAND, G.U.EXPAND:
            a ← REG[ra]
            b ← REG[rb]
        G.COPY, G.SWAP, G.DEAL, G.SHUFFLE:
            a ← REG[ra] || REG[rb]
    endcase
```

```
case op of
    G.ADD:
        for i ← 0 to 128-size by size
            c_{i+size-1..i} ← a_{i+size-1..i} + b_{i+size-1..i}
        endfor
    G.MUL:
        for i ← 0 to 64-size by size
            c_{2*(i+size)-1..2*i} ← (a_{size-1}^{size} || a_{size-1+i..i}) * (b_{size-1}^{size} || b_{size-1+i..i})
        endfor
    G.U.MUL:
        for i ← 0 to 64-size by size
            c_{2*(i+size)-1..2*i} ← (0^{size} || a_{size-1+i..i}) * (0^{size} || b_{size-1+i..i})
        endfor
    G.DIV:
        if (b = 0) or ( (a = (1||0^{63})) and (b = 1^{64}) ) then
            c ← undefined
        else
            q ← a / b
            r ← a - q*b
            c ← r_{63..0} || q_{63..0}
        endif
    G.U.DIV:
        if b = 0 then
            c ← undefined
        else
            q ← (0 || a) / (0 || b)
            r ← a - q*b
            c ← r_{63..0} || q_{63..0}
        endif
    G.AND:
        c ← a and b
    G.OR:
        c ← a or b
    G.XOR:
        c ← a xor b
    G.ANDN:
        c ← a and not b
    G.NAND:
        c ← not (a and b)
    G.NOR:
        c ← not (a or b)
    G.XNOR:
        c ← not (a xor b)
    G.ORN:
        c ← a or not b
    G.POLY:
        p[0] ← a
        for i ← 1 to size
            p[i] ← (p[i-1]_0 ? (0^{64} || b) : 0^{128}) xor (p[i-1]_0 || p[i-1]_{127..1})
        endfor
        c ← p[size]
    G.GATHER:
        for k ← 0 to 128-size by size
            j ← k
            for i ← k to k+size-1 by 1
                if a_i then
                    c_j ← b_i
```

```
                    j ← j + 1
                endif
            endfor
            j ← k+size-1
            for i ← k+size-1 to k by -1
                if ~ai then
                    cj ← bi
                    j ← j - 1
                endif
            endfor
        endfor
    G.SCATTER:
        for k ← 0 to 128-size by size
            j ← k
            for i ← k to k+size-1 by 1
                if ai then
                    ci ← bj
                    j ← j + 1
                endif
            endfor
            j ← k+size-1
            for i ← k+size-1 to k by -1
                if ~ai then
                    cj ← bi
                    j ← j - 1
                endif
            endfor
        endfor
    G.COMPRESS:
        for i ← 0 to 64-size by size
            ci+size-1..i ← ai+i+size-1+(b&(size-1))..i+(b&(size-1))
        endfor
    G.EXPAND:
        for i ← 0 to 64-size by size
            ci+i+size+size-1..i+i ← a i+size-1 size-(b&(size-1)) || ai+size-1..i || 0 b&(size-1)
        endfor
    G.U.EXPAND:
        for i ← 0 to 64-size by size
            ci+i+size+size-1..i+i ← 0 size-(b&(size-1)) || ai+size-1..i || 0 b&(size-1)
        endfor
    G.SHL:
        for i ← 0 to 128-size by size
            ci+size-1..i ← ai+size-1-(b&(size-1))..i || 0 b&(size-1)
        endfor
    G.SHR:
        for i ← 0 to 128-size by size
            ci+size-1..i ← ai+size-1 b&(size-1) || ai+size-1..i+(b&(size-1))
        endfor
    G.U.SHR:
        for i ← 0 to 128-size by size
            ci+size-1..i ← 0 b&(size-1) || ai+size-1..i+(b&(size-1))
        endfor
    G.COPY:
        for i ← 0 to 128-size by size
            ci+size-1..i ← asize-1..0
```

```
            endfor
        G.SWAP:
            for i ← 0 to 128-size by size
                c[i+size-1..i] ← a[127-i..128-size-i]
            endfor
        G.DEAL:
            for i ← 0 to 128-size by size
                j ← (i[5..0] || 0[1])+(i[6] ? size : 0)
                c[i+size-1..i] ← a[j+size-1..j]
            endfor
        G.SHUFFLE:
            for i ← 0 to 128-size by size
                j ← (0[1] || i[6..1])+((i&size) ? (64-(0[1] || size[6..1])) : 0)
                c[i+size-1..i] ← a[j+size-1..j]
            endfor
    endcase
    REG[rc] ← c
enddef
```

Exceptions

Reserved Instruction